EECS 214 Worksheet Solutions: Hash Tables

Question 1. For each of the following inputs *x*, compute the resulting hash value *h(x)*.

1. `"aaa"` → 11 + 11 + 11 = 33 % 10 = **3**
2. `"abc"` → 11 + 22 + 33 = 66 % 10 = **6**
3. `"bbc"` → 22 + 22 + 33 = 77 % 10 = **7**
4. `"bcb"` → 22 + 33 + 22 = 77 % 10 = **7**
5. `"acb"` → 11 + 33 + 22 = 66 % 10 = **6**
6. `"cac"` → 33 + 11 + 33 = 77 % 10 = **7**

Question 2.
If we were using **chaining**, what would the hash table look like after we inserted all 6 strings? (Draw out your linked lists if necessary.)

- **0:** `"cac"` (collision, +3 and modular wrap-around)
- **1:**
- **2:**
- **3:** `"aaa"`
- **4:**
- **5:**
- **6:** `"abc"`
- **7:** `"bbc"`
- **8:** `"bcb"` (collision, +1)
- **9:** `"acb"` (collision, +2)

What would the hash table look like if we were using **open addressing with linear probing**?

- **0:**
- **1:**
- **2:**
- **3:** `"aaa"`
- **4:**
- **5:**
- **6:** `"abc"` → `"acb"` → `"cac"`
- **7:** `"bbc"` → `"bcb"`
- **8:**

- **9:**

Question 3.

1. What is the load factor alpha of the chained hash table? **6/10 = 0.6**

2. What is the load factor alpha of the open addressed hash table? **6/10 = 0.6**

3. Suppose after entering the above values into our open addressed hash table, we wanted to *look up* the string `"cac"`.
   a. How many iterations would it take to find the index of this string? **4 (compute hash function to get 7, check next 3 indices)**
   b. What is the *t*ime complexity of this particular lookup operation? **O(1 + 0.6) = O(1)**
   c. How might it change with 21 entries? **More collisions → closer to O(n)**

Question 4.

1. What are some **disadvantages** of linear probing? **Collisions**
   a. How might we mitigate these disadvantages? **Quadratic probing or double hashing**

2. What are some **disadvantages** of chaining? **Linked lists can grow long**

Sample Interview Questions:

> *These are all questions that we have personally encountered on interviews. While there are other ways to solve these questions besides using hash tables, solutions using hash tables will have strong time complexities for these questions. Knowing how to use hash tables effectively will give you great returns on the computer science job hunt! (Some of us have used hash tables on interviews more times than we'd like to count)*
> *Note that we are not necessarily distinguishing between a chained hash table and an open addressed hash table for these problems.*

Use psuedocode/Python/C# code to solve the following problems

Question 5.

Using a hash table, find whether two inputted strings, *s1*, and *s2*, are valid anagrams of each other. Return true if true, and false otherwise. What would the time complexity be of this algorithm? If we didn't use a hash table, and instead simply iterated through each character of a string with all the characters of the other string, what would the time complexity be?

> *Note that an anagram occurs when the two strings have the same exact number of ocurrences for the same characters. I.e. falafel and llaaeff are anagrams of each other, but falafel and laef are not.*

Solution:
-Create hash table with key being chars in the string and values being the integer number of occurrences
-Check if strings are the same size, return false otherwise
-Iterate through first string, adding the char to the hash table if it isn't there with a value of 1, and if its there with a value of 1, increment that value.
-iterate through the second string, looking up each char along the way in the hash table. If the char isn't there, immediately return false. If it is, decrement its value in the hash table (unless there is a value of 1, in which case delete it from the hash table). If you reach the end of the seconds string, then you know that since both strings are the same length, they must be anagrams of each other, return true

Question 6.
Using a hash table, find whether an array contains at least one set of duplicate elements. What would the time complexity be of this algorithm? If we didn't use a hash table, and instead simply iterated through each character of a string with all the characters of the other string, what would the time complexity be?

> *Hint: try using a hash table which has a value of boolean type.*
> *An example input → output would be*
> *{1, 2, 3, 4, 5, 3} -> true,*
> *{1, 2, 3} -> false*
> *{1, 1, 2, 2 } -> true*

Solution:
-Create hash table with key being the same type as in the array, and value being a boolean.
-Iterate through array, checking if each element is in the hash table. If the element is not, add it, with a value of true. If the element is in the hash table, return true, you have a duplicate. If you reach end of the array, then you know the answer is false as there are no more potential elements.

Question 7.

Solution:
-Create has table with key being type in the array and value being the number of times it appears.
-Then store the length of the floor(variable/2) +1 in a variable that we will call majority.
-Iterate through the array, looking up the element as you go in the hash table. If the element isn't there, add it with a value of 1. If it is there, check if the value is equal to majority-1. If it is then you know this element would make it a majority so you can return true. Otherwise, add one to the value of the element in the hash table. If you reach the end, then you know there was no majority, so return false.